EXHIBIT 1

U.S. Patent No. 7,519,814 ("'814 Patent")

Accused Instrumentalities: Amazon products and services using secure containerized applications, including without limitation Amazon AWS Elastic Container Service (ECS) (including ECS Anywhere), AWS Elastic Kubernetes Service (EKS) (including EKS Anywhere), AWS EC2 (including spot instances), AWS Elastic Container Registry (ECR), and AWS App2Container, and all versions and variations thereof since the issuance of the asserted patent.

Each Accused Instrumentality, separately and in conjunction with each other as described below, infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

Claim 1

Claim 1	Accused Instrumentalities
[1pre] 1. In a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a	To the extent the preamble is limiting, Amazon and/or its customer practices, through the Accused Instrumentalities, in a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, as claimed. For example, AWS ECS, AWS EKS, and AWS EC2 each operates within a system having individual servers, each of which uses an independent operating system. Amazon provides and/or requires that each server includes a processor with one or more cores available to the OS kernel. Amazon further provides and/or requires that each server has a supported operating system (*e.g.*, Amazon Linux 2023, Amazon Linux 2, Amazon EKS Distro, Bottlerocket, Windows Server 2019, Ubuntu, Red Hat Enterprise Linux), which includes a kernel and associated local system files, including for example libraries such as libc/glibc, configuration files, etc. On information and belief, there exist at least two

Claim 1	Accused Instrumentalities
secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, the method comprising:	AWS servers that have different operating systems, for example Amazon Linux 2023 and Amazon Linux 2, or different versions thereof. *See* claim limitations below. *See also*, *e.g.*: What is Amazon Elastic Container Service (ECS)? Amazon ECS is a fully managed opinionated container orchestration service that delivers the easiest way for organizations to build, deploy, and manage containerized applications at any scale on AWS, in traditional Amazon Elastic Cloud Compute (EC2) instances or on a serverless compute plane with AWS Fargate. Amazon ECS is fully managed and versionless, providing tooling and built-in support that makes it simple to build and run containerized applications on AWS. For example, Amazon ECS Service Connect simplifies service discovery, connectivity, and traffic observability while Amazon CloudWatch Container Insights collects, aggregates, and summarizes metrics and logs. With Amazon ECS, you do not have to provision or scale servers or clusters or choose the types of severs you want your containers to run on or optimize cluster packing. You retain control of the operating properties of containers with the ability to specify CPU and memory requirements, networking and IAM policies, and launch type and data volumes. With simple API calls, you can launch and stop container-enabled applications, query the complete state of your cluster, and access many familiar features like security groups, Elastic Load Balancing (ELB), Amazon Elastic Block Store (EBS) volumes, and Identity Access Management (IAM) roles. You can use Amazon ECS to schedule container placement across your cluster based on your resource needs and availability requirements. https://aws.amazon.com/ecs/faqs/

Claim 1	Accused Instrumentalities
	Q: What is Amazon Elastic Kubernetes Service (Amazon EKS)? A: Amazon EKS is a managed service that makes it easy for you to run Kubernetes on AWS without installing and operating your own Kubernetes control plane or worker nodes. **Q: What is Kubernetes?** A: Kubernetes is an open-source container orchestration system allowing you to deploy and manage containerized applications at scale. Kubernetes arranges containers into logical groupings for management and discoverability, then launches them onto clusters of Amazon Elastic Compute Cloud (Amazon EC2) instances. Using Kubernetes, you can run containerized applications including microservices, batch processing workers, and platforms as a service (PaaS) using the same toolset on premises and in the cloud. **Q: Why should I use Amazon EKS?** A: Amazon EKS provisions and scales the Kubernetes control plane, including the application programming interface (API) servers and backend persistence layer, across multiple AWS Availability Zones (AZs) for high availability and fault tolerance. Amazon EKS automatically detects and replaces unhealthy control plane nodes and patches the control plane. You can run EKS using AWS Fargate, which provides serverless compute for containers. Fargate removes the need to provision and manage servers, lets you specify and pay for resources per application, and improves security through application isolation by design. Amazon EKS is integrated with many AWS services to provide scalability and security for your applications. These services include Elastic Load Balancing for load distribution, AWS Identity and Access Management (IAM) for authentication, Amazon Virtual Private Cloud (VPC) for isolation, and AWS CloudTrail for logging. https://aws.amazon.com/eks/faqs/

Claim 1	Accused Instrumentalities
	Q: What is Amazon Elastic Compute Cloud (Amazon EC2)? Amazon EC2 is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. **Q: What can I do with Amazon EC2?** Just as Amazon Simple Storage Service (Amazon S3) enables storage in the cloud, Amazon EC2 enables "compute" in the cloud. The Amazon EC2 simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. https://aws.amazon.com/ec2/faqs/ Amazon Ads chose Amazon Web Services (AWS) to reduce time spent managing infrastructure, lower costs, and optimize ad selection by choosing from the broadest and deepest selection of compute and machine learning capabilities to meet its latency and performance requirements. Using Amazon Elastic Container Service (Amazon ECS) and AWS App Mesh, Amazon Ads built a micro-service inferencing architecture, which scaled model hosting and optimized hardware and software optimizations for each type of inference model. The company chose NVIDIA Triton Inference Servers running on GPU-based Amazon Elastic Compute Cloud (Amazon EC2) G4dn instances for ultra-low latency predictions with deep neutral networks. For asynchronous predictions using BERT models, Amazon Ads uses Amazon SageMaker Multi-Model Endpoints running on Amazon EC2 Inf1 instances, which deliver 2.3x higher throughput and up to 70 percent lower cost per inference than comparable current generation GPU-based Amazon EC2 instances. https://aws.amazon.com/solutions/case-studies/amazonads-kunliu-video-case-study/?did=cr_card&trk=cr_card

Claim 1	Accused Instrumentalities
	• **Container** – Containers provide a standard way to package your application's code, configurations, and dependencies into a single object. Containers share an operating system installed on the host server and run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless of environment. A container is a runnable instance of an image (see Deep Dive on Containers on AWS getting started resource center). While a container image is immutable, the container adds a read/write layer on top of the image to which your application can write information like temporary files or logs (see Docker overview). When the container stops and is removed, the information written to the temporary read/write layer will be lost. • **Container image** – Container images are read-only templates used to build out containers. Container images are immutable, meaning they cannot be changed once created. Container images are created using layers by reading a text file that is called a Dockerfile that contains all necessary information. You can find the Dockerfile reference here. • **Container runtime** – Software running on a container host (virtual machine or bare metal server) operating system that is responsible for running and managing containers. The container relies on the kernel of the host for all system calls. • **Dockerfile** – A file or series of files containing commands that describe the content of a container image. Each command represents a layer on the Container image (see the *Container image layers* definition). • **Container build** – A container image built from a Dockerfile. This process results in a container image containing the necessary components to run your containerized application. • **Base image** – A starting point container image that is used in the container build process to generate custom or new container images. This image has `FROM scratch` in the Dockerfile. https://docs.aws.amazon.com/wellarchitected/latest/container-build-lens/container-technology-terminology.html

Claim 1	Accused Instrumentalities
	Container technology uses the resource-isolation features of the Linux kernel to sandbox an application, its dependencies, configuration files, and interfaces inside an atomic unit called a container. This allows a container to run on any host with the suitable kernel components, while shielding the application from behavioral inconsistencies through variances in software installed on the host. Containers use operating system (OS) level virtualization compared to VMs, which use hardware level virtualization using hypervisor. A hypervisor is a software or a firmware that creates and runs VMs. Multiple containers can run on a single host OS without needing a hypervisor, while isolated from neighboring containers. This layer of isolation allows consistency, flexibility, and portability, which enable rapid software deployment and testing. There are many ways in which using containers on AWS can benefit your organization. Containers have been widely employed in use cases such as distributed applications, batch jobs, and continuous deployment pipelines. The use cases for containers continue to grow in areas like distributed data processing, streaming media delivery, genomics, and machine learning, including generative AI. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html

Claim 1	Accused Instrumentalities
	Amazon ECS-optimized Linux AMIs PDF | RSS Amazon ECS provides the Amazon ECS-optimized AMIs that are preconfigured with the requirements and recommendations to run your container workloads. We recommend that you use the Amazon ECS-optimized Amazon Linux 2023 AMI for your Amazon EC2 instances unless your application requires Amazon EC2 GPU-based instances, a specific operating system or a Docker version that is not yet available in that AMI. For information about the Amazon Linux 2 and Amazon Linux 2023 instances, see Comparing Amazon Linux 2 and *Amazon Linux 2023* in the *Amazon Linux 2023 User Guide*. Launching your container instances from the most recent Amazon ECS-Optimized AMI ensures that you receive the current security updates and container agent version. For information about how to launch an instance, see Launching an Amazon ECS Linux container instance. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_AMI.html; *see also* list of differences between Amazon Linux 2 and Amazon Linux 2023 at https://docs.aws.amazon.com/linux/al2023/ug/compare-with-al2.html

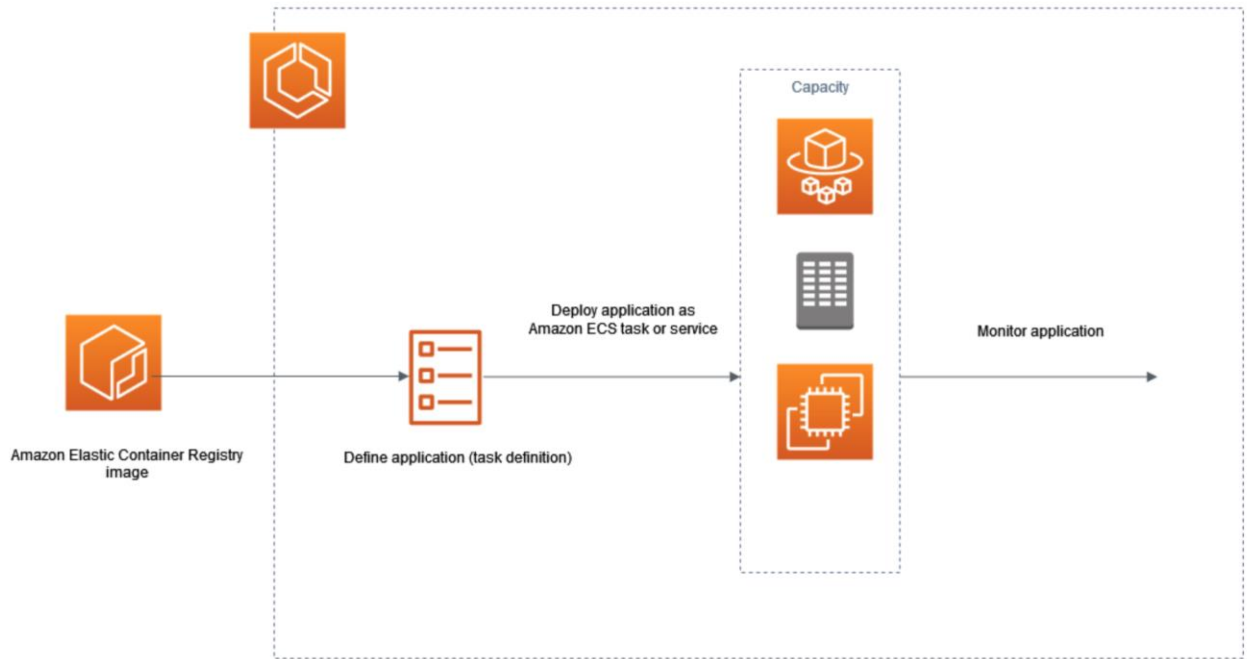
Claim 1	Accused Instrumentalities
	Amazon ECS-optimized Bottlerocket AMIs PDF | RSS Bottlerocket is a Linux based open-source operating system that is purpose built by AWS for running containers on virtual machines or bare metal hosts. The Amazon ECS-optimized Bottlerocket AMI is secure and only includes the minimum number of packages that's required to run containers. This improves resource usage, reduces security attack surface, and helps lower management overhead. The Bottlerocket AMI is also integrated with Amazon ECS to help reduce the operational overhead involved in updating container instances in a cluster. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-bottlerocket.html **Amazon ECS-optimized Windows AMIs** PDF | RSS The Amazon ECS-optimized AMIs are preconfigured with the necessary components that you need to run Amazon ECS workloads. Although you can create your own container instance AMI that meets the basic specifications needed to run your containerized workloads on Amazon ECS, the Amazon ECS-optimized AMIs are preconfigured and tested on Amazon ECS by AWS engineers. It is the simplest way for you to get started and to get your containers running on AWS quickly. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_windows_AMI.html

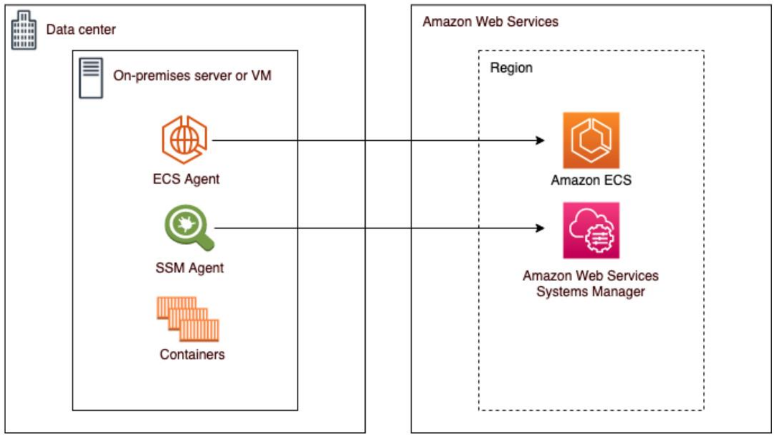
Claim 1	Accused Instrumentalities
	Amazon EKS optimized Amazon Linux AMIs PDF | RSS The Amazon EKS optimized Amazon Linux AMI is built on top of Amazon Linux 2 (AL2) and Amazon Linux 2023 (AL2023). It's configured to serve as the base image for Amazon EKS nodes. The AMI is configured to work with Amazon EKS and it includes the following components: • `kubelet` • AWS IAM Authenticator • Docker (Amazon EKS version `1.23` and earlier) • `containerd` https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-ami.html Amazon EKS optimized Bottlerocket AMIs PDF | RSS Bottlerocket is an open source Linux distribution that's sponsored and supported by AWS. Bottlerocket is purpose-built for hosting container workloads. With Bottlerocket, you can improve the availability of containerized deployments and reduce operational costs by automating updates to your container infrastructure. Bottlerocket includes only the essential software to run containers, which improves resource usage, reduces security threats, and lowers management overhead. The Bottlerocket AMI includes `containerd`, `kubelet`, and AWS IAM Authenticator. In addition to managed node groups and self-managed nodes, Bottlerocket is also supported by Karpenter. https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-ami-bottlerocket.html

Claim 1	Accused Instrumentalities
	# Amazon EKS optimized Ubuntu Linux AMIs PDF | RSS Canonical has partnered with Amazon EKS to create node AMIs that you can use in your clusters. Canonical delivers a built-for-purpose Kubernetes Node OS image. This minimized Ubuntu image is optimized for Amazon EKS and includes the custom AWS kernel that is jointly developed with AWS. For more information, see Ubuntu on Amazon Elastic Kubernetes Service (EKS) and Launching self-managed Ubuntu nodes . For information about support, see the Third-party software section of the *AWS Premium Support FAQs*. https://docs.aws.amazon.com/eks/latest/userguide/eks-partner-amis.html

Claim 1	Accused Instrumentalities
	# Amazon EKS optimized Windows AMIs PDF | RSS Windows Amazon EKS optimized AMIs are built on top of Windows Server 2019 and Windows Server 2022. They are configured to serve as the base image for Amazon EKS nodes. By default, the AMIs include the following components: • kubelet • kube-proxy • AWS IAM Authenticator for Kubernetes • csi-proxy • containerd **Note** You can track security or privacy events for Windows Server with the Microsoft security update guide. Amazon EKS offers AMIs that are optimized for Windows containers in the following variants: • Amazon EKS-optimized Windows Server 2019 Core AMI • Amazon EKS-optimized Windows Server 2019 Full AMI • Amazon EKS-optimized Windows Server 2022 Core AMI • Amazon EKS-optimized Windows Server 2022 Full AMI https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-windows-ami.html

Claim 1	Accused Instrumentalities
	Q: Which operating systems does Amazon EKS support? A: Amazon EKS supports Kubernetes-compatible Linux x86, ARM, and Windows Server operating system distributions. Amazon EKS provides optimized AMIs for Amazon Linux 2, Bottlerocket, and Windows Server 2019. At this time, there is no Amazon EKS optimized AMI for AL2023. EKS- optimized AMIs for other Linux distributions, such as Ubuntu, are available from their respective vendors. https://aws.amazon.com/eks/faqs/ **Q: What infrastructure and operating systems can I use with Amazon EKS Anywhere?** Amazon EKS Anywhere supports different types of infrastructure including VMWare vSphere, bare metal, AWS Snowball Edge, Apache CloudStack, and Nutanix. Amazon EKS Anywhere provides Bottlerocket, a Linux-based open-source operating system built by AWS, as the default node operating system. You can alternatively use Ubuntu and Red Hat Enterprise Linux (RHEL) as the node operating system. https://aws.amazon.com/eks/eks-anywhere/faqs/

Claim 1	Accused Instrumentalities
	Amazon ECS Application Lifecycle  https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

Claim 1	Accused Instrumentalities
	# Amazon ECS clusters for the external launch type PDF | RSS Amazon ECS Anywhere provides support for registering an *external instance* such as an on-premises server or virtual machine (VM), to your Amazon ECS cluster. External instances are optimized for running applications that generate outbound traffic or process data. If your application requires inbound traffic, the lack of Elastic Load Balancing support makes running these workloads less efficient. Amazon ECS added a new `EXTERNAL` launch type that you can use to create services or run tasks on your external instances. The following provides a high-level system architecture overview of Amazon ECS Anywhere. Your on-premises server has both the Amazon ECS agent and the SSM agent installed.  https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-anywhere.html

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Supported operating systems and system architectures

The following is the list of supported operating systems and system architectures.

- Amazon Linux 2
- CentOS 7
- RHEL 7, RHEL 8 — Neither Docker or RHEL's open package repositories support installing Docker natively on RHEL. You must ensure that Docker is installed before you run the install script that's described in this document.
- Fedora 32, Fedora 33
- openSUSE Tumbleweed
- Ubuntu 18, Ubuntu 20, Ubuntu 22
- Debian 10

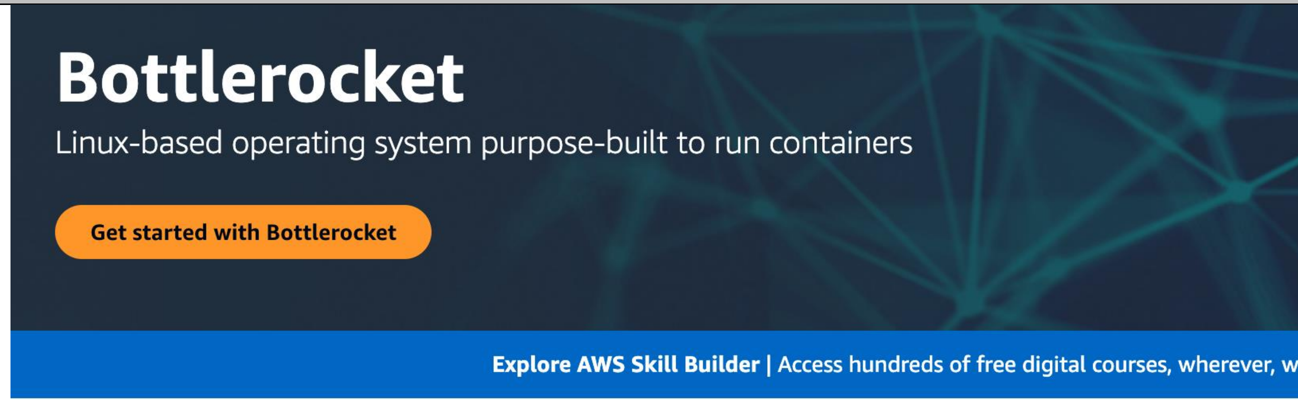
Important

Debian 9 Long Term Support (LTS support) ended on June 30, 2022 and is no longer supported by Amazon ECS Anywhere.

- Debian 11
- Debian 12 — The NVIDIA Container Toolkit isn't currently supported on Debian 12. You won't be able to run GPUs on Debian 12 instances.
- SUSE Enterprise Server 15
- The `x86_64` and `ARM64` CPU architectures are supported.
- The following Windows operating system versions are supported:
  - Windows Server 2022
  - Windows Server 2019
  - Windows Server 2016
  - Windows Server 20H2

https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-anywhere.html

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	 Bottlerocket is a Linux-based open-source operating system that is purpose-built by Amazon Web Services for running containers. Bottlerocket includes only the essential software required to run containers, and ensures that the underlying software is always secure. With Bottlerocket, customers can reduce maintenance overhead and automate their workflows by applying configuration settings consistently as nodes are upgraded or replaced. Bottlerocket is now generally available at no cost as an Amazon Machine Image (AMI) for Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/bottlerocket/
[1a] storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a	The method practiced by Amazon and/or its customer through the Accused Instrumentalities includes a step of storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers. For example, AWS ECS, AWS EKS, and AWS EC2 each stores application containers, sometimes called Docker containers, container images, Kubernetes containers, or Kubernetes pods, in persistent

Claim 1	Accused Instrumentalities
set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers;	storage available to each node running the application. The container might be in a format defined by the Open Container Initiative. This storage may be physically attached to the server or connected through any supported interconnect, including over a network, either within the AWS environment or on-premises. Each container includes the application software as well as a Linux or Windows user space required to execute the application, for example libc/glibc and other shared libraries (or Windows equivalents, such as DLLs), configuration files, etc. necessary for the application. For example, the container includes a base OS image, provided by Amazon or by a third party, such as an Amazon Linux, Alpine Linux, Debian, Ubuntu, or Windows Server base image. The container is compatible with the host kernel, for example because the container libraries are linked against the Linux kernel, and the supported host operating systems also use the Linux kernel, which has a stable binary interface. Likewise, Windows applications are compiled to work with Windows, which has a stable binary interface. For another example, AWS ECS, AWS EKS, and AWS EC2 each stores files, pertaining to the applications, in ephemeral or persistent volumes, required to execute the applications within those containers. Because these volumes are stored and accessible within the runtime environment, it is inferred that they are stored in the memory of the server as claimed. *See, e.g.*:

Claim 1	Accused Instrumentalities
	# Architect your application for Amazon ECS PDF | RSS You architect your application by creating a task definition for your application. The task definition contains the parameters that define information about the application, including: • The launch type to use, which determines the infrastructure that your tasks are hosted on. When you use the EC2 launch type, you also choose the instance type. For some instance types, such as GPU, you need to set additional parameters. For more information, see Amazon ECS task definition use cases. • The container image, which holds your application code and all the dependencies that your application code requires to run. • The networking mode to use for the containers in your task The networking mode determines how your task communicates over the network. For tasks that run on EC2 instance, there are multiple options, but we recommend that you use the `awsvpc` network mode. The `awsvpc` network mode simplifies container networking, because you have more control over how your applications communicate with each other and other services within your VPCs. For tasks that run on Fargate, you can only use the `awsvpc` network mode. • The logging configuration to use for your tasks. • Any data volumes that are used with the containers in the task. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/application_architecture.html

Claim 1	Accused Instrumentalities
	Container image type PDF | RSS The time that it takes a container to start up varies, based on the underlying container image. For example, a fatter image (full versions of Debian, Ubuntu, and Amazon1/2) might take longer to start up because there are a more services that run in the containers compared to their respective slim versions (Debian-slim, Ubuntu-slim, and Amazon-slim) or smaller base images (Alpine). https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/container-type.html • **Container** – Containers provide a standard way to package your application's code, configurations, and dependencies into a single object. Containers share an operating system installed on the host server and run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless of environment. A container is a runnable instance of an image (see Deep Dive on Containers on AWS getting started resource center). While a container image is immutable, the container adds a read/write layer on top of the image to which your application can write information like temporary files or logs (see Docker overview). When the container stops and is removed, the information written to the temporary read/write layer will be lost. • **Container image** – Container images are read-only templates used to build out containers. Container images are immutable, meaning they cannot be changed once created. Container images are created using layers by reading a text file that is called a Dockerfile that contains all necessary information. You can find the Dockerfile reference here. • **Container runtime** – Software running on a container host (virtual machine or bare metal server) operating system that is responsible for running and managing containers. The container relies on the kernel of the host for all system calls. • **Dockerfile** – A file or series of files containing commands that describe the content of a container image. Each command represents a layer on the Container image (see the *Container image layers* definition). • **Container build** – A container image built from a Dockerfile. This process results in a container image containing the necessary components to run your containerized application. • **Base image** – A starting point container image that is used in the container build process to generate custom or new container images. This image has `FROM scratch` in the Dockerfile. https://docs.aws.amazon.com/wellarchitected/latest/container-build-lens/container-technology-terminology.html

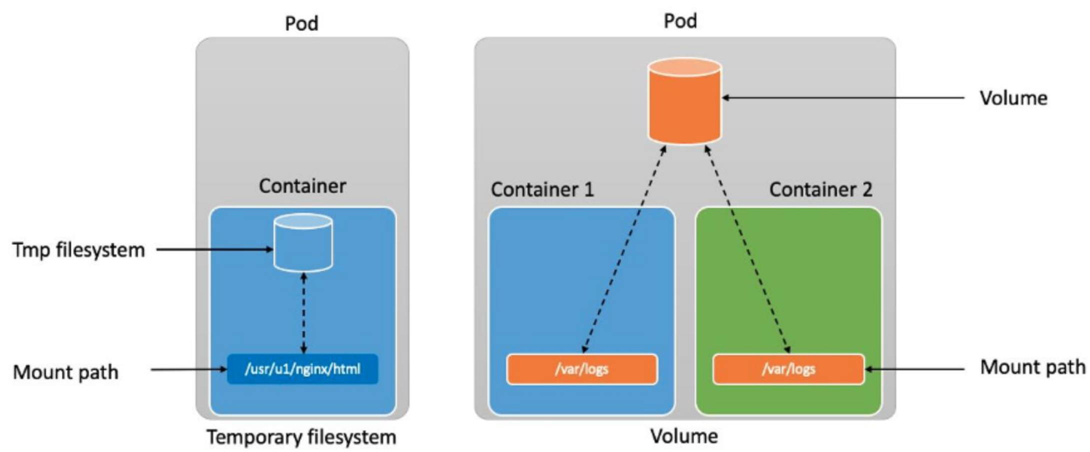
Claim 1	Accused Instrumentalities
	Base image The base image is the selected image and operating system used in your image or container recipe document, along with the components. The base image and the component definitions combined produce the desired configuration for the output image. https://docs.aws.amazon.com/imagebuilder/latest/userguide/what-is-image-builder.html Using a base image from a trusted source can improve the security and reliability of your container. This is because you can be confident that the base image has been thoroughly tested and vetted. It can also reduce the burden of establishing provenance, because you only need to consider the packages and libraries that you include in your image, rather than the entire base image. Here is an example creating a Dockerfile using the official Python base image from the Amazon ECR repository. In fact, all of the Docker official images are available on Amazon ECR public gallery. https://aws.amazon.com/blogs/containers/building-better-container-images/

Claim 1	Accused Instrumentalities
	Docker Official Images now available on Amazon Elastic Container Registry Public by Saleem Muhammad | on 29 NOV 2021 | in Amazon Elastic Container Registry, Announcements, Containers | Permalink | Developers building container-based applications can now discover and download Docker Official Images directly from Amazon Elastic Container Registry (Amazon ECR) Public. This new capability gives AWS customers a simple and highly available way to pull Docker Official Images, while taking advantage of the generous AWS Free Tier. Customers pulling images from Amazon ECR Public to any AWS Region get virtually unlimited downloads. For workloads running outside of AWS, users not authenticated on AWS receive 500 GB of data downloads each month. For additional data downloads, they can sign up or sign in to an AWS account to get up to 5TB of data downloads each month after which they pay $0.09 per GB. Docker Official Images are a curated set of container images published by Docker. Some examples of these images include base OS (for example Ubuntu and CentOS), databases (for example MySQL and Redis), and sidecars that are the starting point for container-based applications. Until today, customers using Docker Official Images could only find these images on Docker Hub, a container registry hosted by Docker. https://aws.amazon.com/blogs/containers/docker-official-images-now-available-on-amazon-elastic-container-registry-public/

Claim 1	Accused Instrumentalities
	Create a Docker image Amazon ECS task definitions use Docker images to launch containers on the container instances in your clusters. In this section, you create a Docker image of a simple web application, and test it on your local system or Amazon EC2 instance, and then push the image to the Amazon ECR container registry so you can use it in an Amazon ECS task definition. **To create a Docker image of a simple web application** 1. Create a file called `Dockerfile`. A Dockerfile is a manifest that describes the base image to use for your Docker image and what you want installed and running on it. For more information about Dockerfiles, go to the Dockerfile Reference. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-container-image.html Using the AL2023 base container image PDF | RSS The AL2023 container image is built from the same software components that are included in the AL2023 AMI. It's available for use in any environment as a base image for Docker workloads. If you're using the Amazon Linux AMI for applications in Amazon Elastic Compute Cloud (Amazon EC2), you can containerize your applications with the Amazon Linux container image. https://docs.aws.amazon.com/linux/al2023/ug/base-container.html

Claim 1	Accused Instrumentalities
	# Pulling the Amazon Linux container image PDF | RSS The Amazon Linux container image is built from the same software components that are included in the Amazon Linux AMI. The Amazon Linux container image is available for use in any environment as a base image for Docker workloads. If you use the Amazon Linux AMI for applications in Amazon EC2, you can containerize your applications with the Amazon Linux container image. You can use the Amazon Linux container image in your local development environment and then push your application to AWS using Amazon ECS. For more information, see Using Amazon ECR images with Amazon ECS. https://docs.aws.amazon.com/AmazonECR/latest/userguide/amazon_linux_container_image.html

Claim 1	Accused Instrumentalities
	# Store application data PDF | RSS This chapter covers storage options for Amazon EKS clusters. **Topics** • Use Amazon EBS storage • Use Amazon EFS storage • Use Amazon FSx for Lustre storage • Use Amazon FSx for NetApp ONTAP storage • Use Amazon FSx for OpenZFS storage • Use Amazon File Cache • Use Mountpoint for Amazon S3 storage • Use snapshot controller with CSI storage https://docs.aws.amazon.com/eks/latest/userguide/storage.html **Persistent storage for Kubernetes** by Suman Debnath, Daniel Rubinstein, Anjani Reddy, and Narayana Vemburaj | on 22 NOV 2022 | in Advanced (300), Amazon Elastic File System (EFS), Amazon Elastic Kubernetes Service, Technical How-to | Permalink | Comments | Stateful applications rely on data being persisted and retrieved to run properly. When running stateful applications using Kubernetes, state needs to be persisted regardless of container, pod, or node crashes or terminations. This requires persistent storage, that is, storage that lives beyond the lifetime of the container, pod, or node. https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/

Claim 1	Accused Instrumentalities
	Kubernetes volumes Kubernetes has several types of storage options available, not all of which are persistent. Ephemeral storage Containers can use the `temporary filesystem` (tmpfs) to read and write files. However, ephemeral storage does not satisfy the three storage requirements. In case of a container crash, the `temporary filesystem` is lost—the container starts with a clean slate again. Also, multiple containers cannot share a `temporary filesystem`. Ephemeral volumes An ephemeral Kubernetes `Volume` solves both of the problems faced with ephemeral storage. An ephemeral `Volume` 's lifetime is coupled to the `Pod`. It enables safe container restarts and sharing of data between containers within a `Pod`. However as soon as the `Pod` is deleted, the `Volume` is deleted as well, so it still does not fulfill our three requirements.  *The temporary file system is tied to the lifecycle of the container; the ephemeral Volume is tied to the lifecycle of the pod* https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/

Claim 1	Accused Instrumentalities
	Decoupling pods from the storage: Persistent Volumes Kubernetes also supports `Persistent Volumes`. With `Persistent Volumes`, data is persisted regardless of the lifecycle of the application, container, Pod, Node, or even the cluster itself. `Persistent Volumes` fulfill the three requirements outlined earlier. A `Persistent Volume (PV)` object represents a storage volume that is used to persist application data. A `PV` has its own lifecycle, separate from the lifecycle of Kubernetes `Pods`. A `PV` essentially consists of two different things: • A backend technology called a `PersistentVolume` • An access mode, which tells Kubernetes how the volume should be mounted. **Backend technology** A `PV` is an abstract component, and the actual physical storage must come from somewhere. Here are a few examples: • `csi`: Container Storage Interface (CSI) → (for example, Amazon EFS, Amazon EBS, Amazon FSx, etc.) • `iscsi`: iSCSI (SCSI over IP) storage • `local`: Local storage devices mounted on nodes • `nfs`: Network File System (NFS) storage https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/

Claim 1	Accused Instrumentalities
	Container Storage Interface (CSI) drivers The Container Storage Interface (CSI) is an abstraction designed to facilitate using different storage solutions with Kubernetes. Different storage vendors can develop their own drivers that implement the CSI standards, enabling their storage solutions to work with Kubernetes (regardless of the internals of the underlying storage solution). AWS has CSI plugins for Amazon EBS, Amazon EFS , and Amazon FSx for Lustre. https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/ 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/ At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the `.spec.containers[*].volumeMounts`. A process in a container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes mount at the specified paths within the image. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	What is containerization? Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine. However, with containerization, you can create a single software package, or container, that runs on all types of devices and operating systems. https://aws.amazon.com/what-is/containerization/ **How does containerization work?** Containerization involves building self-sufficient software packages that perform consistently, regardless of the machines they run on. Software developers create and deploy container images—that is, files that contain the necessary information to run a containerized application. Developers use containerization tools to build container images based on the Open Container Initiative (OCI) image specification. OCI is an open-source group that provides a standardized format for creating container images. Container images are read-only and cannot be altered by the computer system. Container images are the top layer in a containerized system that consists of the following layers. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	Infrastructure Infrastructure is the hardware layer of the container model. It refers to the physical computer or bare-metal server that runs the containerized application. **Operating system** The second layer of the containerization architecture is the operating system. Linux is a popular operating system for containerization with on-premise computers. In cloud computing, developers use cloud services such as AWS EC2 to run containerized applications. **Container engine** The container engine, or container runtime, is a software program that creates containers based on the container images. It acts as an intermediary agent between the containers and the operating system, providing and managing resources that the application needs. For example, container engines can manage multiple containers on the same operating system by keeping them independent of the underlying infrastructure and each other. **Application and dependencies** The topmost layer of the containerization architecture is the application code and the other files it needs to run, such as library dependencies and related configuration files. This layer might also contain a light guest operating system that gets installed over the host operating system. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	What are the types of container technology? The following are some examples of popular technologies that developers use for containerization. **Docker** Docker, or Docker Engine, is a popular open-source container runtime that allows software developers to build, deploy, and test containerized applications on various platforms. Docker containers are self-contained packages of applications and related files that are created with the Docker framework. **Linux** Linux is an open-source operating system with built-in container technology. Linux containers are self-contained environments that allow multiple Linux-based applications to run on a single host machine. Software developers use Linux containers to deploy applications that write or read large amounts of data. Linux containers do not copy the entire operating system to their virtualized environment. Instead, the containers consist of necessary functionalities allocated in the Linux namespace. **Kubernetes** Kubernetes is a popular open-source container orchestrator that software developers use to deploy, scale, and manage a vast number of microservices. It has a declarative model that makes automating containers easier. The declarative model ensures that Kubernetes takes the appropriate action to fulfil the requirements based on the configuration files. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	How Docker containers work A Docker container is a runtime environment with all the necessary components—like code, dependencies, and libraries—needed to run the application code without using host machine dependencies. This container runtime runs on the engine on a server, machine, or cloud instance. The engine runs multiple containers depending on the underlying resources available. To deploy and scale a set of containers to communicate effectively across different machines or virtual machines, you need a container orchestration platform like Kubernetes. This helps whether your machines are on premises or in the cloud. Kubernetes manages multiple machines, known as a cluster, within the context of container operations. Read about Kubernetes » **How Docker images work** A Docker image, or container image, is a standalone, executable file used to create a container. This container image contains all the libraries, dependencies, and files that the container needs to run. A Docker image is shareable and portable, so you can deploy the same image in multiple locations at once—much like a software binary file. You can store images in registries to keep track of complex software architectures, projects, business segments, and user group access. For instance, the public Docker Hub registry contains images such as operating systems, programming language frameworks, databases, and code editors. https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/

Claim 1	Accused Instrumentalities
	About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

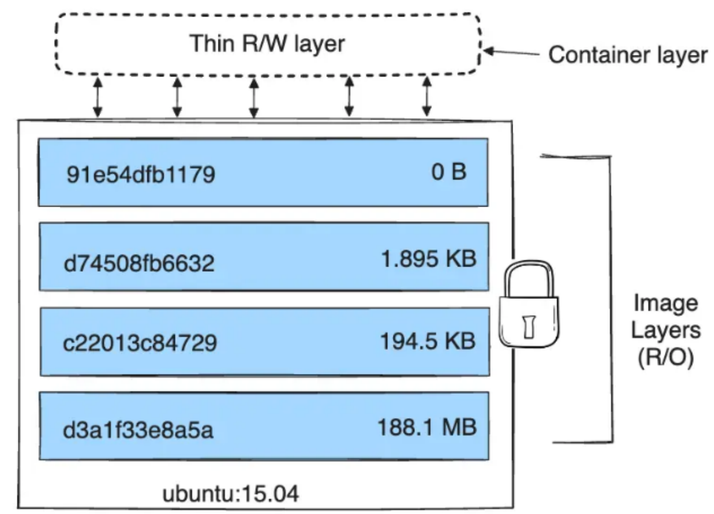
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

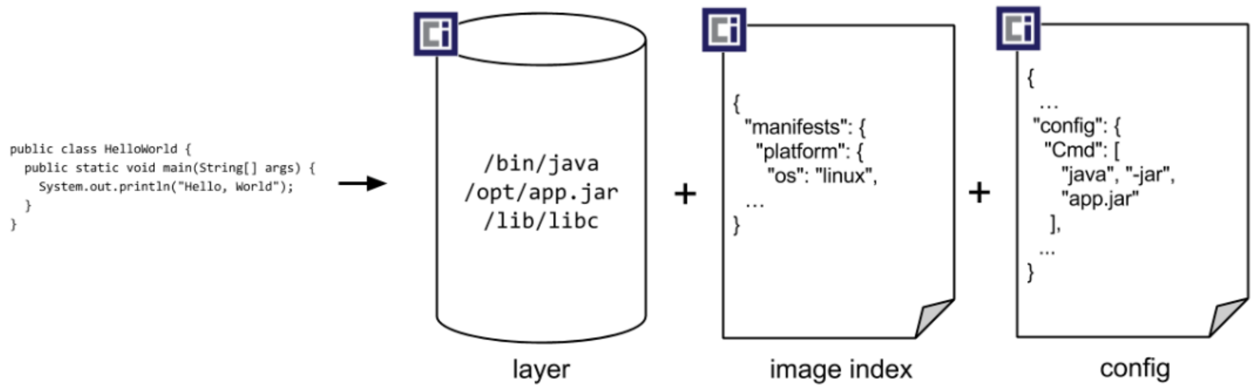
Claim 1	Accused Instrumentalities
	Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	Portability The flexibility of containers is based on its portability, ease of deployment, and smaller size compared to virtual machines. The Open Container Initiative (OCI), was formed to support fully interoperable container open standards with 3 specifications: • The Runtime Specification (runtime-spec) • The Image Specification (image-spec) • The Distribution Specification (distribution-spec) The OCI image specification defines an OCI Image, consisting of an image manifest, which contains metadata about contents and dependencies of the image; an image index (optional); a set of filesystem layers; and a configuration, such as arguments and environment variables. You can run the OCI compliant container image on any supported version of Linux or Windows, if you have the OCI compliant container runtime installed on the host. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/container-benefits.html The open container initiative (OCI) images generated by docker build tools will continue to run in your Amazon EKS clusters as before. As an end-user of Kubernetes, you will not experience significant changes. You can continue to use Docker to build your containers outside the cluster. Visit this link to learn why Amazon EKS is discontinuing Dockershim support. For more information, see Kubernetes is Moving on From Dockershim: Commitments and Next Steps on the *Kubernetes Blog*. https://aws.amazon.com/blogs/containers/amazon-eks-now-supports-kubernetes-version-1-24/

Claim 1	Accused Instrumentalities
	A container runtime, also known as container engine, is a software component that can run containers on a host operating system. Container runtimes are responsible for loading container images from a repository, monitoring local system resources, isolating system resources for use of a container, and managing container lifecycle. They come in two forms: • High-level container runtimes (such as *containerd* and *CRI-O*) provide functions that run on top of low-level runtime. • Low-level runtimes are responsible for creating and running containers. The primary job of the low-level container runtimes is to provide container lifecycle management. These runtimes implement the Runtime Specification provided by the OCI(Open Container Initiative), a Linux Foundation project started by Docker, which aims to provide open standards for Linux containers. The default reference implementation for low level runtimes specified by OCI is *runc*. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/key-considerations.html

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	## Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	• **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ◦ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ◦ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
[1b] wherein the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems,	In the method practiced by Amazon through the Accused Instrumentalities, the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems. The system files in the container are compatible with the host kernel, for example because they are linked against the Linux kernel and the supported host operating systems also use the Linux kernel, which has a stable binary interface. Likewise, Windows applications are compiled to work with Windows, which has a stable binary interface. *See* discussion and evidence in element [1a] above. *See also, e.g.*:

Claim 1	Accused Instrumentalities
	What are the types of container technology? The following are some examples of popular technologies that developers use for containerization. **Docker** Docker, or Docker Engine, is a popular open-source container runtime that allows software developers to build, deploy, and test containerized applications on various platforms. Docker containers are self-contained packages of applications and related files that are created with the Docker framework. **Linux** Linux is an open-source operating system with built-in container technology. Linux containers are self-contained environments that allow multiple Linux-based applications to run on a single host machine. Software developers use Linux containers to deploy applications that write or read large amounts of data. Linux containers do not copy the entire operating system to their virtualized environment. Instead, the containers consist of necessary functionalities allocated in the Linux namespace. **Kubernetes** Kubernetes is a popular open-source container orchestrator that software developers use to deploy, scale, and manage a vast number of microservices. It has a declarative model that makes automating containers easier. The declarative model ensures that Kubernetes takes the appropriate action to fulfil the requirements based on the configuration files. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	How Docker containers work A Docker container is a runtime environment with all the necessary components—like code, dependencies, and libraries—needed to run the application code without using host machine dependencies. This container runtime runs on the engine on a server, machine, or cloud instance. The engine runs multiple containers depending on the underlying resources available. To deploy and scale a set of containers to communicate effectively across different machines or virtual machines, you need a container orchestration platform like Kubernetes. This helps whether your machines are on premises or in the cloud. Kubernetes manages multiple machines, known as a cluster, within the context of container operations. Read about Kubernetes » **How Docker images work** A Docker image, or container image, is a standalone, executable file used to create a container. This container image contains all the libraries, dependencies, and files that the container needs to run. A Docker image is shareable and portable, so you can deploy the same image in multiple locations at once—much like a software binary file. You can store images in registries to keep track of complex software architectures, projects, business segments, and user group access. For instance, the public Docker Hub registry contains images such as operating systems, programming language frameworks, databases, and code editors. https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/ **Q: What container images can I run in containers on Bottlerocket?** Bottlerocket can run all container images that meet the OCI Image Format specification and Docker images. **Q: Can I move my containers running on Amazon Linux 2 to Bottlerocket?** Yes, you can move your containers across Amazon Linux 2 and Bottlerocket without modifications. https://aws.amazon.com/bottlerocket/faqs/

Claim 1	Accused Instrumentalities
	Container technology uses the resource-isolation features of the Linux kernel to sandbox an application, its dependencies, configuration files, and interfaces inside an atomic unit called a container. This allows a container to run on any host with the suitable kernel components, while shielding the application from behavioral inconsistencies through variances in software installed on the host. Containers use operating system (OS) level virtualization compared to VMs, which use hardware level virtualization using hypervisor. A hypervisor is a software or a firmware that creates and runs VMs. Multiple containers can run on a single host OS without needing a hypervisor, while isolated from neighboring containers. This layer of isolation allows consistency, flexibility, and portability, which enable rapid software deployment and testing. There are many ways in which using containers on AWS can benefit your organization. Containers have been widely employed in use cases such as distributed applications, batch jobs, and continuous deployment pipelines. The use cases for containers continue to grow in areas like distributed data processing, streaming media delivery, genomics, and machine learning, including generative AI. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html
[1c] the containers of application software excluding a kernel,	In the method practiced by Amazon and/or its customer through the Accused Instrumentalities, the containers of application software exclude a kernel. *See* discussion and evidence in element [1a] above. *See also, e.g.*: **Summary of difference: Docker vs. VM** | | **Docker container** | **VM** | | Architecture | Shares resources with the underlying host kernel. | Runs its own kernel and operating system. | https://aws.amazon.com/compare/the-difference-between-docker-vm/

Claim 1	Accused Instrumentalities
	Docker containers, on the other hand, use resources on demand. Rather than asking for a specific amount of physical hardware resourcing as virtual machines do, they simply request what they need from the single operating system kernel. Multiple containers share the same operating system. Docker containers direct resource sharing with the kernel leads and may use less system resources compared to a VM. **Security** Because Docker containers share the kernel with the host operating system, for lightweight resource consumption, they're at risk if there are vulnerabilities in the kernel. However, Docker also provides many advanced security controls. https://aws.amazon.com/compare/the-difference-between-docker-vm/

Claim 1	Accused Instrumentalities
	(table below)

Criteria	EKS managed node groups	Self managed nodes
Can be deployed to AWS Outposts	No	Yes
Can be deployed to an AWS Local Zone	No	Yes – For more information, see Amazon EKS and AWS Local Zones.
Can run containers that require Windows	Yes	Yes – Your cluster still requires at least one (two recommended for availability) Linux node though.
Can run containers that require Linux	Yes	Yes
Can run workloads that require the Inferentia chip	Yes – Amazon Linux nodes only	Yes – Amazon Linux only
Can run workloads that require a GPU	Yes – Amazon Linux nodes only	Yes – Amazon Linux only
Can run workloads that require Arm processors	Yes	Yes
Can run AWS Bottlerocket	Yes	Yes
Pods share a kernel runtime environment with other Pods	Yes – All of your Pods on each of your nodes	Yes – All of your Pods on each of your nodes
Pods share CPU, memory, storage, and network resources with other Pods.	Yes – Can result in unused resources on each node	Yes – Can result in unused resources on each node

https://docs.aws.amazon.com/eks/latest/userguide/eks-compute.html

Claim 1	Accused Instrumentalities
	• **Container runtime** – Software running on a container host (virtual machine or bare metal server) operating system that is responsible for running and managing containers. The container relies on the kernel of the host for all system calls. https://docs.aws.amazon.com/wellarchitected/latest/container-build-lens/container-technology-terminology.html 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/
[1d] wherein some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server,	In the method practiced by Amazon and/or its customer through the Accused Instrumentalities, some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server. For example, each container will utilize its own local system files, including libraries such as libc/glibc (or Windows equivalents, such as DLLs) and configuration files, not the corresponding libraries and configuration files of the host OS. *See* discussion and evidence in element [1a] above. *See also, e.g.*:

Claim 1	Accused Instrumentalities
	# Architect your application for Amazon ECS PDF | RSS You architect your application by creating a task definition for your application. The task definition contains the parameters that define information about the application, including: • The launch type to use, which determines the infrastructure that your tasks are hosted on. When you use the EC2 launch type, you also choose the instance type. For some instance types, such as GPU, you need to set additional parameters. For more information, see Amazon ECS task definition use cases. • The container image, which holds your application code and all the dependencies that your application code requires to run. • The networking mode to use for the containers in your task The networking mode determines how your task communicates over the network. For tasks that run on EC2 instance, there are multiple options, but we recommend that you use the `awsvpc` network mode. The `awsvpc` network mode simplifies container networking, because you have more control over how your applications communicate with each other and other services within your VPCs. For tasks that run on Fargate, you can only use the `awsvpc` network mode. • The logging configuration to use for your tasks. • Any data volumes that are used with the containers in the task. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/application_architecture.html

Claim 1	Accused Instrumentalities
	# Container image type PDF | RSS The time that it takes a container to start up varies, based on the underlying container image. For example, a fatter image (full versions of Debian, Ubuntu, and Amazon1/2) might take longer to start up because there are a more services that run in the containers compared to their respective slim versions (Debian-slim, Ubuntu-slim, and Amazon-slim) or smaller base images (Alpine). https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/container-type.html • **Container** – Containers provide a standard way to package your application's code, configurations, and dependencies into a single object. Containers share an operating system installed on the host server and run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless of environment. A container is a runnable instance of an image (see Deep Dive on Containers on AWS getting started resource center). While a container image is immutable, the container adds a read/write layer on top of the image to which your application can write information like temporary files or logs (see Docker overview). When the container stops and is removed, the information written to the temporary read/write layer will be lost. • **Container image** – Container images are read-only templates used to build out containers. Container images are immutable, meaning they cannot be changed once created. Container images are created using layers by reading a text file that is called a Dockerfile that contains all necessary information. You can find the Dockerfile reference here. • **Container runtime** – Software running on a container host (virtual machine or bare metal server) operating system that is responsible for running and managing containers. The container relies on the kernel of the host for all system calls. • **Dockerfile** – A file or series of files containing commands that describe the content of a container image. Each command represents a layer on the Container image (see the *Container image layers* definition). • **Container build** – A container image built from a Dockerfile. This process results in a container image containing the necessary components to run your containerized application. • **Base image** – A starting point container image that is used in the container build process to generate custom or new container images. This image has `FROM scratch` in the Dockerfile. https://docs.aws.amazon.com/wellarchitected/latest/container-build-lens/container-technology-terminology.html

Claim 1	Accused Instrumentalities
	Base image The base image is the selected image and operating system used in your image or container recipe document, along with the components. The base image and the component definitions combined produce the desired configuration for the output image. https://docs.aws.amazon.com/imagebuilder/latest/userguide/what-is-image-builder.html **How does containerization work?** Containerization involves building self-sufficient software packages that perform consistently, regardless of the machines they run on. Software developers create and deploy container images—that is, files that contain the necessary information to run a containerized application. Developers use containerization tools to build container images based on the Open Container Initiative (OCI) image specification. OCI is an open-source group that provides a standardized format for creating container images. Container images are read-only and cannot be altered by the computer system. Container images are the top layer in a containerized system that consists of the following layers. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	Infrastructure Infrastructure is the hardware layer of the container model. It refers to the physical computer or bare-metal server that runs the containerized application. **Operating system** The second layer of the containerization architecture is the operating system. Linux is a popular operating system for containerization with on-premise computers. In cloud computing, developers use cloud services such as AWS EC2 to run containerized applications. **Container engine** The container engine, or container runtime, is a software program that creates containers based on the container images. It acts as an intermediary agent between the containers and the operating system, providing and managing resources that the application needs. For example, container engines can manage multiple containers on the same operating system by keeping them independent of the underlying infrastructure and each other. **Application and dependencies** The topmost layer of the containerization architecture is the application code and the other files it needs to run, such as library dependencies and related configuration files. This layer might also contain a light guest operating system that gets installed over the host operating system. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	What are the types of container technology? The following are some examples of popular technologies that developers use for containerization. **Docker** Docker, or Docker Engine, is a popular open-source container runtime that allows software developers to build, deploy, and test containerized applications on various platforms. Docker containers are self-contained packages of applications and related files that are created with the Docker framework. **Linux** Linux is an open-source operating system with built-in container technology. Linux containers are self-contained environments that allow multiple Linux-based applications to run on a single host machine. Software developers use Linux containers to deploy applications that write or read large amounts of data. Linux containers do not copy the entire operating system to their virtualized environment. Instead, the containers consist of necessary functionalities allocated in the Linux namespace. **Kubernetes** Kubernetes is a popular open-source container orchestrator that software developers use to deploy, scale, and manage a vast number of microservices. It has a declarative model that makes automating containers easier. The declarative model ensures that Kubernetes takes the appropriate action to fulfil the requirements based on the configuration files. https://aws.amazon.com/what-is/containerization/

Claim 1	Accused Instrumentalities
	How Docker containers work A Docker container is a runtime environment with all the necessary components—like code, dependencies, and libraries—needed to run the application code without using host machine dependencies. This container runtime runs on the engine on a server, machine, or cloud instance. The engine runs multiple containers depending on the underlying resources available. To deploy and scale a set of containers to communicate effectively across different machines or virtual machines, you need a container orchestration platform like Kubernetes. This helps whether your machines are on premises or in the cloud. Kubernetes manages multiple machines, known as a cluster, within the context of container operations. Read about Kubernetes » **How Docker images work** A Docker image, or container image, is a standalone, executable file used to create a container. This container image contains all the libraries, dependencies, and files that the container needs to run. A Docker image is shareable and portable, so you can deploy the same image in multiple locations at once—much like a software binary file. You can store images in registries to keep track of complex software architectures, projects, business segments, and user group access. For instance, the public Docker Hub registry contains images such as operating systems, programming language frameworks, databases, and code editors. https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/
[1e] wherein said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server,	In the method practiced by Amazon and/or its customer through the Accused Instrumentalities, said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server. For example, in some cases the host OS and container will use one or more identical system files, for example when both the host and the container incorporate the same Linux distribution version, or when both host and container use the same version of libc (*e.g.*, where both host and container use the same version of Amazon Linux). In other cases modified copies are used instead, for example when different versions of the same library, or configuration files with different parameters, are used by the host and container.

Claim 1	Accused Instrumentalities
	See discussion and evidence in element [1a] above. *See also, e.g.*: **How Docker containers work** A Docker container is a runtime environment with all the necessary components—like code, dependencies, and libraries—needed to run the application code without using host machine dependencies. This container runtime runs on the engine on a server, machine, or cloud instance. The engine runs multiple containers depending on the underlying resources available. To deploy and scale a set of containers to communicate effectively across different machines or virtual machines, you need a container orchestration platform like Kubernetes. This helps whether your machines are on premises or in the cloud. Kubernetes manages multiple machines, known as a cluster, within the context of container operations. Read about Kubernetes » **How Docker images work** A Docker image, or container image, is a standalone, executable file used to create a container. This container image contains all the libraries, dependencies, and files that the container needs to run. A Docker image is shareable and portable, so you can deploy the same image in multiple locations at once—much like a software binary file. You can store images in registries to keep track of complex software architectures, projects, business segments, and user group access. For instance, the public Docker Hub registry contains images such as operating systems, programming language frameworks, databases, and code editors. https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/

Claim 1	Accused Instrumentalities
	# Amazon ECS-optimized Linux AMIs PDF | RSS Amazon ECS provides the Amazon ECS-optimized AMIs that are preconfigured with the requirements and recommendations to run your container workloads. We recommend that you use the Amazon ECS-optimized Amazon Linux 2023 AMI for your Amazon EC2 instances unless your application requires Amazon EC2 GPU-based instances, a specific operating system or a Docker version that is not yet available in that AMI. For information about the Amazon Linux 2 and Amazon Linux 2023 instances, see Comparing Amazon Linux 2 and *Amazon Linux 2023* in the *Amazon Linux 2023 User Guide*. Launching your container instances from the most recent Amazon ECS-Optimized AMI ensures that you receive the current security updates and container agent version. For information about how to launch an instance, see Launching an Amazon ECS Linux container instance. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_AMI.html # Using the AL2023 base container image PDF | RSS The AL2023 container image is built from the same software components that are included in the AL2023 AMI. It's available for use in any environment as a base image for Docker workloads. If you're using the Amazon Linux AMI for applications in Amazon Elastic Compute Cloud (Amazon EC2), you can containerize your applications with the Amazon Linux container image. https://docs.aws.amazon.com/linux/al2023/ug/base-container.html

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h1>Pulling the Amazon Linux container image</h1>
PDF | RSS
The Amazon Linux container image is built from the same software components that are included in the Amazon Linux AMI. The Amazon Linux container image is available for use in any environment as a base image for Docker workloads. If you use the Amazon Linux AMI for applications in Amazon EC2, you can containerize your applications with the Amazon Linux container image.
You can use the Amazon Linux container image in your local development environment and then push your application to AWS using Amazon ECS. For more information, see Using Amazon ECR images with Amazon ECS.
https://docs.aws.amazon.com/AmazonECR/latest/userguide/amazon_linux_container_image.html
COPY and ADD: These commands copy files and directories from your local filesystem into the Docker image. They are often used to include your application code, configuration files, and dependencies.
https://medium.com/@swalperen3008/what-is-dockerize-and-dockerize-your-project-a-step-by-step-guide-899c48a34df6
<h2>Container images</h2>
A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings.
https://kubernetes.io/docs/concepts/containers/
</td></tr>
</table>

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

About storage drivers

To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way.

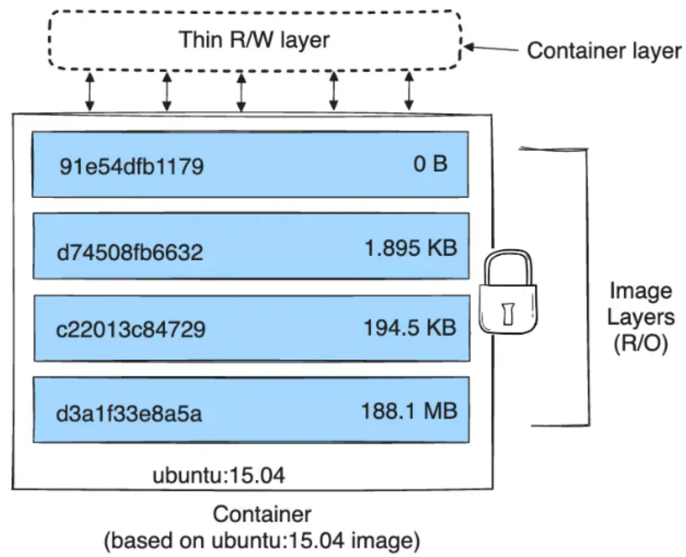
Storage drivers versus Docker volumes

Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer.

Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance.

https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Images and layers A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile: `# syntax=docker/dockerfile:1` `FROM ubuntu:22.04` `LABEL org.opencontainers.image.authors="org@example.com"` `COPY . /app` `RUN make /app` `RUN rm -r $HOME/.cache` `CMD python /app/app.py` This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
[1f] and wherein the application software cannot be shared between the plurality of secure containers of application software,	In the method practiced by Amazon through the Accused Instrumentalities, the application software cannot be shared between the plurality of secure containers of application software. For example, each container has an isolated runtime environment that cannot be accessed by other containers, for example including a per-container writeable layer or other ephemeral per-container storage. For another example, when the plurality of secure containers each corresponds to a different container image, each container cannot access another container's image and therefore application software. *See, e.g.*: Container technology uses the resource-isolation features of the Linux kernel to sandbox an application, its dependencies, configuration files, and interfaces inside an atomic unit called a container. This allows a container to run on any host with the suitable kernel components, while shielding the application from behavioral inconsistencies through variances in software installed on the host. Containers use operating system (OS) level virtualization compared to VMs, which use hardware level virtualization using hypervisor. A hypervisor is a software or a firmware that creates and runs VMs. Multiple containers can run on a single host OS without needing a hypervisor, while isolated from neighboring containers. This layer of isolation allows consistency, flexibility, and portability, which enable rapid software deployment and testing. There are many ways in which using containers on AWS can benefit your organization. Containers have been widely employed in use cases such as distributed applications, batch jobs, and continuous deployment pipelines. The use cases for containers continue to grow in areas like distributed data processing, streaming media delivery, genomics, and machine learning, including generative AI. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html

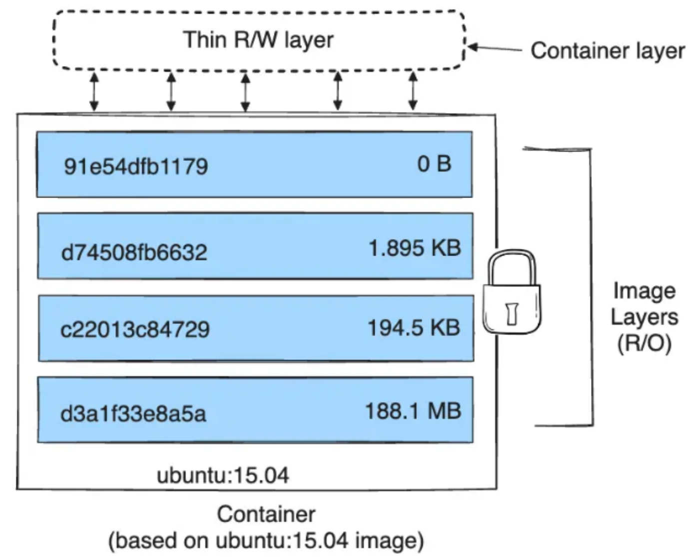
Claim 1	Accused Instrumentalities
	Before we get too deep into technical details, I want to talk about how containers are typically used and why we see some consistent feedback about those themes. In any environment, booting a computer can take a while. But what's harder than booting is deploying a random application to that computer, and doing so reliably. Containers make this process a lot easier. A container image provides a reliable and repeatable mechanism for packaging up the set of local dependencies for an application, including its dynamically linked libraries, other programs to invoke, and assets. The Linux kernel primitives that power containers, including cgroups and namespaces, provide some amount of resource and visibility isolation. Containers also start up much more quickly than a whole computer. These properties enable each application to pretend that it's the only application running, enables subdividing larger computers into smaller parts so more of these applications can run together without conflict, and makes it attractive to use one computer for running multiple applications or even a cluster of computers to run many copies of those applications. https://aws.amazon.com/blogs/containers/bottlerocket-a-special-purpose-container-operating-system/ **Build secure microservices** Ensure strong security isolation between your containers. AWS provides the latest security updates and lets you set granular access permissions for every container. AWS offers over 210 security, compliance, and governance services, plus key features to best suit your needs. https://aws.amazon.com/containers/

Claim 1	Accused Instrumentalities
	Understand Containers offer a number of advantages for packaging, deploying, and running applications: • **Isolation:** Improve security and reliability with containers' process-level isolation, with which applications running in separate containers cannot interfere with each other, improving security and reliability. https://docs.aws.amazon.com/decision-guides/latest/containers-on-aws-how-to-choose/choosing-aws-container-service.html **Fault tolerance** Software development teams use containers to build fault-tolerant applications. They use multiple containers to run microservices on the cloud. Because containerized microservices operate in isolated user spaces, a single faulty container doesn't affect the other containers. This increases the resilience and availability of the application. **Agility** Containerized applications run in isolated computing environments. Software developers can troubleshoot and change the application code without interfering with the operating system, hardware, or other application services. They can shorten software release cycles and work on updates quickly with the container model. https://aws.amazon.com/what-is/containerization/ Linux containers are made up of control groups (cgroups) and namespaces that help limit what a container can access, but all containers share the same Linux kernel as the host Amazon EC2 instance. https://docs.aws.amazon.com/eks/latest/userguide/security.html

Claim 1	Accused Instrumentalities
	Use Windows containers instead of running many applications on one instance of IIS Consider the following advantages of using Windows containers instead of running multiple applications on one EC2 Windows instance with Internet Information Services (IIS): • **Security** – Containers provide a level of security out of the box that isn't achieved through isolation at the IIS level. If one IIS website or application is compromised, all the other hosted sites are exposed and vulnerable. Container escape is rare and a harder vulnerability to exploit than gaining control of a server through a web vulnerability. • **Flexibility** – The ability to run containers in process isolation and have their own instance allows for more granular networking options. Containers also offer complex distribution methods across many EC2 instances. You don't get these benefits when you consolidate applications on a single IIS instance. • **Management overhead** – Server Name Indication (SNI) creates overhead that requires management and automation. Also, you have to grapple with typical operating system management operations like patching, troubleshooting BSOD (if auto scaling isn't in place), endpoint protection, and so on. Configuring IIS sites according to security best practices is a time consuming and ongoing activity. You might even need to set up trust levels, which also adds to management overhead. Containers are designed to be stateless and immutable. Ultimately, your deployments are faster, more secure, and repeatable if you use Windows containers instead. https://docs.aws.amazon.com/prescriptive-guidance/latest/optimize-costs-microsoft-workloads/windows-containers-main.html

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	# Images and layers A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile: `# syntax=docker/dockerfile:1` `FROM ubuntu:22.04` `LABEL org.opencontainers.image.authors="org@example.com"` `COPY . /app` `RUN make /app` `RUN rm -r $HOME/.cache` `CMD python /app/app.py` This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
[1g] and wherein each of the containers has a unique root file system that is different from an operating system's root file system.	In the method practiced by Amazon through the Accused Instrumentalities, each of the containers has a unique root file system that is different from an operating system's root file system. For example, the container's root file system comprises the image layer(s), an ephemeral writeable layer (e.g., in Docker terminology the container layer), and optionally one or more volumes. This root file system is distinct and isolated from the host operating system's root file system. *See, e.g.*: Amazon EC2 instance root volume PDF | RSS When you launch an instance, we create a *root volume* for the instance. The root volume contains the image used to boot the instance. Each instance has a single root volume. You can add storage volumes to your instances during or after launch. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html

Claim 1	Accused Instrumentalities

Storage options for Amazon ECS tasks

PDF | RSS

Amazon ECS provides you with flexible, cost effective, and easy-to-use data storage options depending on yo
needs. Amazon ECS supports the following data volume options for containers:

Data volume	Supported launch types	Supported operating systems	Storage persistence	Use cases
Amazon Elastic Block Store (Amazon EBS)	Fargate, Amazon EC2	Linux	Can be persisted when attached to a standalone task. Ephemeral when attached to a task maintained by a service.	Amazon EBS volumes provide cost-effective, durable, high-performance block storage for data-intensive containerized workloads. Common use cases include transactional workloads such as databases, virtual desktops and root volumes, and throughput intensive workloads such as log processing and ETL workloads. For more information, see Use Amazon EBS volumes with Amazon ECS.

https://docs.aws.amazon.com/AmazonECS/latest/developerguide/using_data_volumes.html

Claim 1	Accused Instrumentalities
	To use a volume, specify the volumes to provide for the Pod in `.spec.volumes` and declare where to mount those volumes into containers in `.spec.containers[*].volumeMounts`. A process in a container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes mount at the specified paths within the image. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

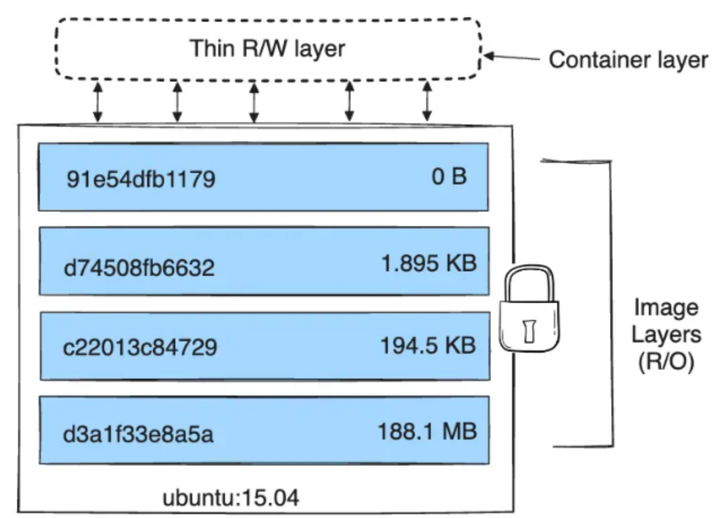
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	The original purpose of the cgroup, **chroot**, and namespace facilities in the kernel was to protect applications from noisy, nosey, and messy neighbors. Combining these with container images created an abstraction that also isolates applications from the (heterogeneous) operating systems https://kubernetes.io/docs/concepts/storage/volumes/ **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

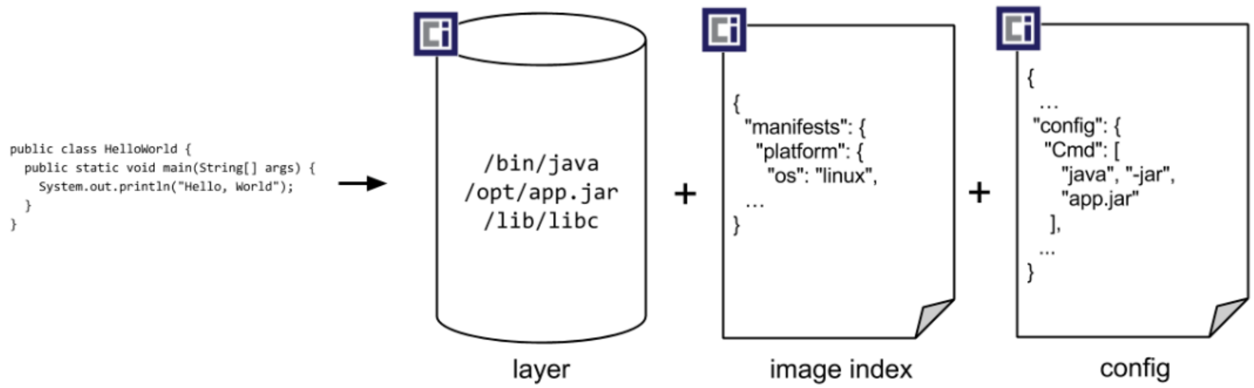
Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	# Open Container Initiative ## Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Overview

At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.



https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

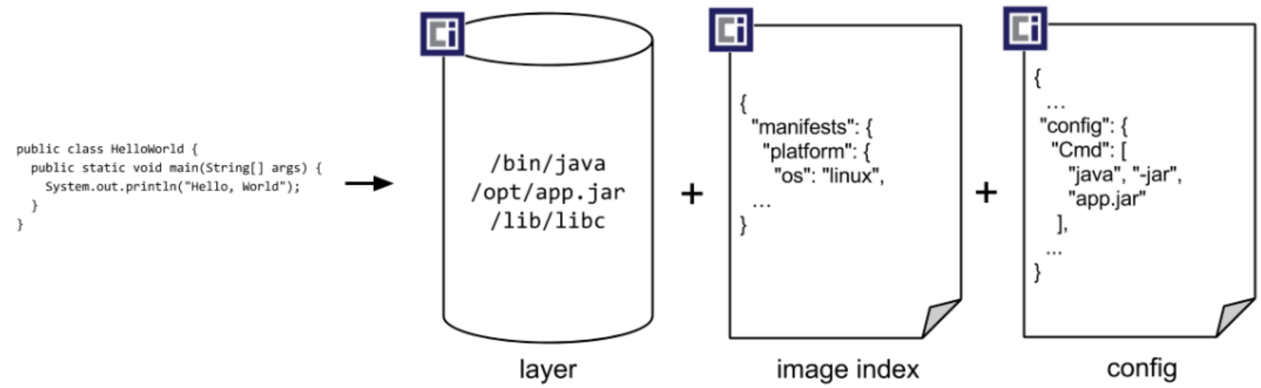
Claim 1	Accused Instrumentalities
	Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	• **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ○ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ○ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 2

Claim 2	Accused Instrumentalities
2. A method as defined in claim 1, wherein each container has an execution file associated therewith for starting the one or more applications.	Amazon and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1, wherein each container has an execution file associated therewith for starting the one or more applications. For example, a container image has an associated image configuration comprising information for starting the one or more applications. This can be an Open Containers Initiative image configuration. *See, e.g.:*

Claim 2	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 2	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 2	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 2	Accused Instrumentalities
	• **config** *object*, OPTIONAL The execution parameters which SHOULD be used as a base when running a container using the image. This field can be `null`, in which case any execution parameters should be specified at creation of the container. ◦ **Env** *array of strings*, OPTIONAL Entries are in the format of `VARNAME=VARVALUE`. These values act as defaults and are merged with any specified when creating a container. ◦ **Entrypoint** *array of strings*, OPTIONAL A list of arguments to use as the command to execute when the container starts. These values act as defaults and may be replaced by an entrypoint specified when creating a container. ◦ **Cmd** *array of strings*, OPTIONAL Default arguments to the entrypoint of the container. These values act as defaults and may be replaced by any specified when creating a container. If an `Entrypoint` value is not specified, then the first entry of the `Cmd` array SHOULD be interpreted as the executable to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 4

Claim 4	Accused Instrumentalities
4. A method as defined in claim 1 further comprising the step of pre-identifying applications and system files required for association with the one or more containers prior to said storing step.	Amazon and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 further comprising the step of pre-identifying applications and system files required for association with the one or more containers prior to said storing step. For example, Amazon's App2Container functionality identifies an application along with its dependencies to be migrated to the target container. This identification step happens before storing the containers having the migrated application and files in the target machine, as described above. *See* analysis and evidence for claim 1 above. *See also, e.g.*: **Q: What is AWS App2Container ?** AWS App2Container (A2C) is a new command line tool that helps transform existing applications running in virtual machines into containers, without needing any code changes. A2C discovers applications running on a server, identifies dependencies, and generates relevant artifacts for seamless deployment to Amazon ECS and EKS. A2C also provides integration with AWS CodeBuild and CodeDeploy to enable a repeatable way to build and deploy containerized application. https://aws.amazon.com/app2container/faqs/

Claim 4	Accused Instrumentalities
	Q: How does App2Container make it easier to containerize applications? App2Container enables you to quickly transform your existing applications running on-premises, in Amazon EC2, or in any other cloud. You get the following benefits by using A2C for containerization: • Application Inventory: A2C identifies the supported ASP.NET and Java applications running in a server which enables a quick and accurate inventory of applications in your environment without extensive manual effort. • Detailed Dependency Analysis: A2C analyzes the running application and identifies dependencies including cooperating processes and network port dependencies. This detailed information reduces the manual effort required to understand and document application anatomy and required dependencies. • Seamless Deployment: A2C generates ECS task definitions and Kubernetes deployment YAML for the containerized application following the AWS best practices for security and scalability by integrating with various AWS services such as ECR, ECS, and EKS. A2C generates CloudFormation template to configure required compute, network, and security infrastructure to seamlessly deploy containerized application in AWS. A2C creates CI/CD pipelines for Amazon DevOps services such as CodeBuild and CodeDeploy to build and deploy containers. If you have existing CI/CD tooling (for example, Azure DevOps, and Jenkins), then you can integrate A2C provided deployment artifacts into your existing CI/CD workflows. https://aws.amazon.com/app2container/faqs/

Claim 4	Accused Instrumentalities
	Q: How do I get started? Once the prerequisites are met and the App2Container tool is installed on the application server, use the following sequence of commands to containerize the application. • Init: Configures the tool with your AWS profile and other related parameters. • Inventory: Identifies all running applications that are supported by A2C. Choose an application to proceed. • Analyze: Performs detailed analysis to identify application artifacts, third-party dependencies, network ports and configuration files, and generates a configuration file to represent the gathered information. You can make additional changes to the configuration file (for example, exclude or include any additional files or directories). • Containerize: Generates a dockerfile and container image for the analyzed application. If needed, you can modify the A2C-generated dockerfile and rebuild the container. You can also perform local testing of the containerized application using docker run command before deploying to AWS container services. • Generate Deployment: Registers the container image with Amazon ECR, creates ECS task definitions, and Kubernetes deployment YAML file to deploy the containerized application to Amazon ECS or Amazon EKS. A2C generates CloudFormation template to create required infrastructure to deploy the containerized application. Optionally, it also generates AWS CodeBuild and CodeDeploy artifacts to enable repeatable way to build and deploy the containerized app. https://aws.amazon.com/app2container/faqs/ **Q: What artifacts are generated by App2Container?** App2Container generates the following artifacts for each application component: 1) Application artifacts such as DLLs configuration, 2) Dockerfile, 3) container image in ECR, 4) ECS Task definitions, 5) Kubernetes deployment YAML, 6) CloudFormation Template, 6) AWS CodeBuild and CodeDeploy resources. https://aws.amazon.com/app2container/faqs/

Claim 6

Claim 6	Accused Instrumentalities
6. A method as defined in claim 2, comprising the step of assigning a unique associated identity to each of	Amazon and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, comprising the step of assigning a unique associated identity to each of a plurality of the containers, wherein the identity includes at least one of IP address, host name, and MAC address.

Claim 6	Accused Instrumentalities
a plurality of the containers, wherein the identity includes at least one of IP address, host name, and MAC address.	For example, Kubernetes containers have an associated hostname, which in the case of a single-container Pod is the unique identity of that container. For another example, Kubernetes pods have an associated hostname, which is unique. For another example, a networked Kubernetes pod has an assigned IPv4 and/or IPv6 address. For another example, a Docker container has an IP address and a hostname. *See, e.g.*: **VPC requirements and considerations** When you create a cluster, the VPC that you specify must meet the following requirements and considerations: • The VPC must have a sufficient number of IP addresses available for the cluster, any nodes, and other Kubernetes resources that you want to create. If the VPC that you want to use doesn't have a sufficient number of IP addresses, try to increase the number of available IP addresses. You can do this by updating the cluster configuration to change which subnets and security groups the cluster uses. You can update from the AWS Management Console, the latest version of the AWS CLI, AWS CloudFormation, and `eksctl` version `v0.164.0-rc.0` or later. You might need to do this to provide subnets with more available IP addresses to successfully upgrade a cluster version. https://docs.aws.amazon.com/eks/latest/userguide/network_reqs.html • If you want Kubernetes to assign `IPv6` addresses to Pods and services, associate an `IPv6` CIDR block with your VPC. For more information, see Associate an `IPv6` CIDR block with your VPC in the Amazon VPC User Guide. https://docs.aws.amazon.com/eks/latest/userguide/network_reqs.html

Claim 6	Accused Instrumentalities
	Kubernetes control plane software decides when and where to run your pods, manages traffic routing, and scales your pods based on utilization or other metrics that you define. Kubernetes automatically starts pods on your cluster based on their resource requirements and automatically restarts pods if they or the instances they are running on fail. Each pod is given an IP address and a single DNS name, which Kubernetes uses to connect your services with each other and external traffic. https://aws.amazon.com/kubernetes/ **Container information** The *hostname* of a Container is the name of the Pod in which the Container is running. It is available through the `hostname` command or the `gethostname` function call in libc. The Pod name and namespace are available as environment variables through the downward API. User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the container image. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 6	Accused Instrumentalities
	IP address and hostname By default, the container gets an IP address for every Docker network it attaches to. A container receives an IP address out of the IP subnet of the network. The Docker daemon performs dynamic subnetting and IP address allocation for containers. Each network also has a default subnet mask and gateway. You can connect a running container to multiple networks, either by passing the `--network` flag multiple times when creating the container, or using the `docker network connect` command for already running containers. In both cases, you can use the `--ip` or `--ip6` flags to specify the container's IP address on that particular network. In the same way, a container's hostname defaults to be the container's ID in Docker. You can override the hostname using `--hostname`. When connecting to an existing network using `docker network connect`, you can use the `--alias` flag to specify an additional network alias for the container on that network. https://docs.docker.com/network/

Claim 8

Claim 8	Accused Instrumentalities
8. A method as defined in claim 1, wherein the one or more applications and associated system files are retrieved from a computer system having a plurality of secure containers.	Amazon and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1, wherein the one or more applications and associated system files are retrieved from a computer system having a plurality of secure containers. For example, AWS EKS, AWS ECS, and AWS E2C can retrieve container images from Amazon's Elastic Container Registry (ECR). *See, e.g.*:

Claim 8	Accused Instrumentalities
	Q: What is Amazon Elastic Container Registry (Amazon ECR)? Amazon ECR is a fully managed container registry that makes it easy for developers to share and deploy container images and artifacts. Amazon ECR is integrated with Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Kubernetes Service (Amazon EKS), and AWS Lambda, simplifying your development to production workflow. Amazon ECR eliminates the need to operate your own container repositories or worry about scaling the underlying infrastructure. Amazon ECR hosts your images in a highly available and scalable architecture, allowing you to deploy containers for your applications reliably. Integration with AWS Identity and Access Management (IAM) provides resource-level control of each repository that lets you share images across your organization or with anyone in the world. https://aws.amazon.com/ecr/faqs/ **Q: Why should I use Amazon ECR?** Amazon ECR eliminates the need to operate and scale the infrastructure required to power your container registry. Amazon ECR uses Amazon Simple Storage Service (S3) for storage to make your container images highly available and accessible, allowing you to deploy new containers for your applications reliably. Amazon ECR transfers your container images over HTTPS and automatically encrypts your images at rest. You can configure policies to manage permissions for each repository and restrict access to IAM users, roles, or other AWS accounts. Amazon ECR integrates with Amazon ECS, Amazon EKS, AWS Fargate, AWS Lambda, and the Docker CLI, allowing you to simplify your development and production workflows. You can easily push your container images to Amazon ECR using the Docker CLI from your development machine, and Amazon container orchestrators or compute can pull them directly for production deployments. https://aws.amazon.com/ecr/faqs/ **Q: How do I pull a public image from Amazon ECR?** You pull using the familiar 'docker pull' command with the URL of the image. You can easily search for this URL by finding images using a publisher alias, image name, or image description using the Amazon ECR public gallery. Image URLs are in the format public.ecr.aws/<alias>/<image>:<tag>, for example public.ecr.aws/eks/aws-alb-ingress-controller:v1.1.5 https://aws.amazon.com/ecr/faqs/

Claim 8	Accused Instrumentalities
	Q: Does Amazon ECR support the Open Container Initiative (OCI) format? Yes. Amazon ECR is compatible with the Open Container Initiative (OCI) image specification, letting you push and pull OCI images and artifacts. Amazon ECR can also translate between Docker Image Manifest V2, Schema 2 images and OCI images on pull. https://aws.amazon.com/ecr/faqs/

Claim 9

Claim 9	Accused Instrumentalities
9. A method as defined in claim 2, wherein server information related to hardware resource usage including at least one of CPU memory, network bandwidth, and disk allocation is associated with at least some of the containers prior to the applications within the containers being executed.	Amazon and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, wherein server information related to hardware resource usage including at least one of CPU memory, network bandwidth, and disk allocation is associated with at least some of the containers prior to the applications within the containers being executed. For example, Kubernetes tracks and limits resource usage, including CPU and memory resources. For another example, Docker tracks and limits resource usage, including CPU and memory resources. *See, e.g.*:

Claim 9	Accused Instrumentalities
	Kubecost Amazon EKS supports Kubecost, which you can use to monitor your costs broken down by Kubernetes resources including Pods, nodes, namespaces, and labels. As a Kubernetes platform administrator and finance leader, you can use Kubecost to visualize a breakdown of Amazon EKS charges, allocate costs, and charge back organizational units such as application teams. You can provide your internal teams and business units with transparent and accurate cost data based on their actual AWS bill. Moreover, you can also get customized recommendations for cost optimization based on their infrastructure environment and usage patterns within their clusters. For more information about Kubecost, see the Kubecost documentation. Amazon EKS provides an AWS optimized bundle of Kubecost for cluster cost visibility. You can use your existing AWS support agreements to obtain support. https://docs.aws.amazon.com/eks/latest/userguide/cost-monitoring.html **Resource Management for Pods and Containers** When you specify a Pod, you can optionally specify how much of each resource a container needs. The most common resources to specify are CPU and memory (RAM); there are others. When you specify the resource *request* for containers in a Pod, the kube-scheduler uses this information to decide which node to place the Pod on. When you specify a resource *limit* for a container, the kubelet enforces those limits so that the running container is not allowed to use more of that resource than the

Claim 9	Accused Instrumentalities
	limit you set. The kubelet also reserves at least the *request* amount of that system resource specifically for that container to use. Requests and limits If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its `request` for that resource specifies. However, a container is not allowed to use more than its resource `limit`. For example, if you set a `memory` request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM. If you set a `memory` limit of 4GiB for that container, the kubelet (and container runtime) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel terminates the process that attempted the allocation, with an out of memory (OOM) error. Limits can be implemented either reactively (the system intervenes once it sees a violation) or by enforcement (the system prevents the container from ever exceeding the limit). Different runtimes can have different ways to implement the same restrictions. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ **Runtime options with Memory, CPUs, and GPUs**

Claim 9	Accused Instrumentalities
	By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows. Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the `docker run` command. This section provides details on when you should set such limits and the possible implications of setting them. **Limit a container's access to memory** Docker can enforce hard or soft memory limits. • Hard limits lets the container use no more than a fixed amount of memory. • Soft limits lets the container use as much memory as it needs unless certain conditions are met, such as when the kernel detects low memory or contention on the host machine. https://docs.docker.com/config/containers/resource_constraints/

Claim 10

Claim 10	Accused Instrumentalities
10. A method as defined in claim 2, wherein in operation when an application residing within a container is executed, said application has no access to system files or applications in other containers or to system files within the operating system during execution thereof.	Amazon and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, wherein in operation when an application residing within a container is executed, said application has no access to system files or applications in other containers or to system files within the operating system during execution thereof. *See, e.g.*:

Claim 10	Accused Instrumentalities
	What is containerization? Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine. However, with containerization, you can create a single software package, or container, that runs on all types of devices and operating systems. https://aws.amazon.com/what-is/containerization/ **How does containerization work?** Containerization involves building self-sufficient software packages that perform consistently, regardless of the machines they run on. Software developers create and deploy container images—that is, files that contain the necessary information to run a containerized application. Developers use containerization tools to build container images based on the Open Container Initiative (OCI) image specification. OCI is an open-source group that provides a standardized format for creating container images. Container images are read-only and cannot be altered by the computer system. Container images are the top layer in a containerized system that consists of the following layers. https://aws.amazon.com/what-is/containerization/

Claim 10	Accused Instrumentalities
	Infrastructure Infrastructure is the hardware layer of the container model. It refers to the physical computer or bare-metal server that runs the containerized application. **Operating system** The second layer of the containerization architecture is the operating system. Linux is a popular operating system for containerization with on-premise computers. In cloud computing, developers use cloud services such as AWS EC2 to run containerized applications. **Container engine** The container engine, or container runtime, is a software program that creates containers based on the container images. It acts as an intermediary agent between the containers and the operating system, providing and managing resources that the application needs. For example, container engines can manage multiple containers on the same operating system by keeping them independent of the underlying infrastructure and each other. **Application and dependencies** The topmost layer of the containerization architecture is the application code and the other files it needs to run, such as library dependencies and related configuration files. This layer might also contain a light guest operating system that gets installed over the host operating system. https://aws.amazon.com/what-is/containerization/

Claim 10	Accused Instrumentalities
	How Docker containers work A Docker container is a runtime environment with all the necessary components—like code, dependencies, and libraries—needed to run the application code without using host machine dependencies. This container runtime runs on the engine on a server, machine, or cloud instance. The engine runs multiple containers depending on the underlying resources available. To deploy and scale a set of containers to communicate effectively across different machines or virtual machines, you need a container orchestration platform like Kubernetes. This helps whether your machines are on premises or in the cloud. Kubernetes manages multiple machines, known as a cluster, within the context of container operations. Read about Kubernetes » **How Docker images work** A Docker image, or container image, is a standalone, executable file used to create a container. This container image contains all the libraries, dependencies, and files that the container needs to run. A Docker image is shareable and portable, so you can deploy the same image in multiple locations at once—much like a software binary file. You can store images in registries to keep track of complex software architectures, projects, business segments, and user group access. For instance, the public Docker Hub registry contains images such as operating systems, programming language frameworks, databases, and code editors. https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/

Claim 10	Accused Instrumentalities
	Container technology uses the resource-isolation features of the Linux kernel to sandbox an application, its dependencies, configuration files, and interfaces inside an atomic unit called a container. This allows a container to run on any host with the suitable kernel components, while shielding the application from behavioral inconsistencies through variances in software installed on the host. Containers use operating system (OS) level virtualization compared to VMs, which use hardware level virtualization using hypervisor. A hypervisor is a software or a firmware that creates and runs VMs. Multiple containers can run on a single host OS without needing a hypervisor, while isolated from neighboring containers. This layer of isolation allows consistency, flexibility, and portability, which enable rapid software deployment and testing. There are many ways in which using containers on AWS can benefit your organization. Containers have been widely employed in use cases such as distributed applications, batch jobs, and continuous deployment pipelines. The use cases for containers continue to grow in areas like distributed data processing, streaming media delivery, genomics, and machine learning, including generative AI. https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html Before we get too deep into technical details, I want to talk about how containers are typically used and why we see some consistent feedback about those themes. In any environment, booting a computer can take a while. But what's harder than booting is deploying a random application to that computer, and doing so reliably. Containers make this process a lot easier. A container image provides a reliable and repeatable mechanism for packaging up the set of local dependencies for an application, including its dynamically linked libraries, other programs to invoke, and assets. The Linux kernel primitives that power containers, including cgroups and namespaces, provide some amount of resource and visibility isolation. Containers also start up much more quickly than a whole computer. These properties enable each application to pretend that it's the only application running, enables subdividing larger computers into smaller parts so more of these applications can run together without conflict, and makes it attractive to use one computer for running multiple applications or even a cluster of computers to run many copies of those applications. https://aws.amazon.com/blogs/containers/bottlerocket-a-special-purpose-container-operating-system/

Claim 10	Accused Instrumentalities
	Build secure microservices Ensure strong security isolation between your containers. AWS provides the latest security updates and lets you set granular access permissions for every container. AWS offers over 210 security, compliance, and governance services, plus key features to best suit your needs. https://aws.amazon.com/containers/ **Understand** Containers offer a number of advantages for packaging, deploying, and running applications: • **Isolation:** Improve security and reliability with containers' process-level isolation, with which applications running in separate containers cannot interfere with each other, improving security and reliability. https://docs.aws.amazon.com/decision-guides/latest/containers-on-aws-how-to-choose/choosing-aws-container-service.html

Claim 10	Accused Instrumentalities
	Fault tolerance Software development teams use containers to build fault-tolerant applications. They use multiple containers to run microservices on the cloud. Because containerized microservices operate in isolated user spaces, a single faulty container doesn't affect the other containers. This increases the resilience and availability of the application. **Agility** Containerized applications run in isolated computing environments. Software developers can troubleshoot and change the application code without interfering with the operating system, hardware, or other application services. They can shorten software release cycles and work on updates quickly with the container model. https://aws.amazon.com/what-is/containerization/ Linux containers are made up of control groups (cgroups) and namespaces that help limit what a container can access, but all containers share the same Linux kernel as the host Amazon EC2 instance. https://docs.aws.amazon.com/eks/latest/userguide/security.html

Claim 10	Accused Instrumentalities
	Use Windows containers instead of running many applications on one instance of IIS Consider the following advantages of using Windows containers instead of running multiple applications on one EC2 Windows instance with Internet Information Services (IIS): • **Security** – Containers provide a level of security out of the box that isn't achieved through isolation at the IIS level. If one IIS website or application is compromised, all the other hosted sites are exposed and vulnerable. Container escape is rare and a harder vulnerability to exploit than gaining control of a server through a web vulnerability. • **Flexibility** – The ability to run containers in process isolation and have their own instance allows for more granular networking options. Containers also offer complex distribution methods across many EC2 instances. You don't get these benefits when you consolidate applications on a single IIS instance. • **Management overhead** – Server Name Indication (SNI) creates overhead that requires management and automation. Also, you have to grapple with typical operating system management operations like patching, troubleshooting BSOD (if auto scaling isn't in place), endpoint protection, and so on. Configuring IIS sites according to security best practices is a time consuming and ongoing activity. You might even need to set up trust levels, which also adds to management overhead. Containers are designed to be stateless and immutable. Ultimately, your deployments are faster, more secure, and repeatable if you use Windows containers instead. https://docs.aws.amazon.com/prescriptive-guidance/latest/optimize-costs-microsoft-workloads/windows-containers-main.html

Claim 13

Claim 13	Accused Instrumentalities
13. A method as defined in claim 1 further comprising the step of associating with a plurality of containers a stored history of when	Amazon and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 further comprising the step of associating with a plurality of containers a stored history of when processes related to applications within the container are executed for at least one of, tracking statistics, resource allocation, and for monitoring the status of the application.

Claim 13	Accused Instrumentalities
processes related to applications within the container are executed for at least one of, tracking statistics, resource allocation, and for monitoring the status of the application.	*See* analysis and evidence for claim 1 above. *See also, e.g.*:

Claim 13	Accused Instrumentalities
	# Logging and Monitoring in Amazon Elastic Container Service PDF | RSS Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon Elastic Container Service and your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. AWS provides several tools for monitoring your Amazon ECS resources and responding to potential incidents: **Amazon CloudWatch Alarms** Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see Monitor Amazon ECS using CloudWatch . For services with tasks that use the Fargate launch type, you can use CloudWatch alarms to scale in and scale out the tasks in your service based on CloudWatch metrics, such as CPU and memory utilization. For more information, see Automatically scale your Amazon ECS service. For clusters with tasks or services using the EC2 launch type, you can use CloudWatch alarms to scale in and scale out the container instances based on CloudWatch metrics, such as cluster memory reservation. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-logging-monitoring.html

Claim 13	Accused Instrumentalities
	Amazon CloudWatch Logs Monitor, store, and access the log files from the containers in your Amazon ECS tasks by specifying the `awslogs` log driver in your task definitions. For more information, see Using the awslogs driver. You can also monitor, store, and access the operating system and Amazon ECS container agent log files from your Amazon ECS container instances. This method for accessing logs can be used for containers using the EC2 launch type.. **Amazon CloudWatch Events** Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see Automate responses to Amazon ECS errors using EventBridge in this guide and What Is Amazon CloudWatch Events? in the *Amazon CloudWatch Events User Guide*. **AWS CloudTrail Logs** CloudTrail provides a record of actions taken by a user, role, or an AWS service in Amazon ECS. Using the information collected by CloudTrail, you can determine the request that was made to Amazon ECS, the IP address from which the request was made, who made the request, when it was made, and additional details. For more information, see Log Amazon ECS API calls using AWS CloudTrail. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-logging-monitoring.html

Claim 13	Accused Instrumentalities
	AWS Compute Optimizer AWS Compute Optimizer is a service that analyzes the configuration and utilization metrics of your AWS resources. It reports whether your resources are optimal, and generates optimization recommendations to reduce the cost and improve the performance of your workloads. For more information, see AWS Compute Optimizer recommendations for Amazon ECS. Another important part of monitoring Amazon ECS involves manually monitoring those items that the CloudWatch alarms don't cover. The CloudWatch, Trusted Advisor, and other AWS console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on your container instances and the containers in your tasks. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-logging-monitoring.html

Claim 14

Claim 14	Accused Instrumentalities
14. A method as defined in claim 1 comprising the step of creating containers prior to said step of storing containers in memory, wherein containers are created by: a) running an instance of a service on a server;	Amazon and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 comprising the step of creating containers prior to said step of storing containers in memory, wherein containers are created by (a) running an instance of a service on a server; (b) determining which files are being used; and, (c) copying applications and associated system files to memory without overwriting the associated system files so as to provide a second instance of the applications and associated system files. For example, AWS App2Container, used in conjunction with ECS, EKS, and/or EC2, supports the creation of containers and deploying the containers on the server(s). The containers are first created

Claim 14	Accused Instrumentalities
b) determining which files are being used; and, c) copying applications and associated system files to memory without overwriting the associated system files so as to provide a second instance of the applications and associated system files.	and then later deployed/stored on the server. The creation step involves determining which applications and files are to be migrated, copying these identified applications and files to a location in the target server. Based on information and belief, once the files are migrated, the earlier stored files (if any) are not deleted/overwritten, rather, the migrated files are stored as different instance in memory accessible to containers. Further, an instance of an application/service may be tested or run on the target server to ensure compatibility. *See* analysis and evidence for claims 1 and 4 above. *See also, e.g.*: **Q: What is AWS App2Container ?** AWS App2Container (A2C) is a new command line tool that helps transform existing applications running in virtual machines into containers, without needing any code changes. A2C discovers applications running on a server, identifies dependencies, and generates relevant artifacts for seamless deployment to Amazon ECS and EKS. A2C also provides integration with AWS CodeBuild and CodeDeploy to enable a repeatable way to build and deploy containerized application. https://aws.amazon.com/app2container/faqs/

Claim 14	Accused Instrumentalities
	Q: How does App2Container make it easier to containerize applications? App2Container enables you to quickly transform your existing applications running on-premises, in Amazon EC2, or in any other cloud. You get the following benefits by using A2C for containerization: • Application Inventory: A2C identifies the supported ASP.NET and Java applications running in a server which enables a quick and accurate inventory of applications in your environment without extensive manual effort. • Detailed Dependency Analysis: A2C analyzes the running application and identifies dependencies including cooperating processes and network port dependencies. This detailed information reduces the manual effort required to understand and document application anatomy and required dependencies. • Seamless Deployment: A2C generates ECS task definitions and Kubernetes deployment YAML for the containerized application following the AWS best practices for security and scalability by integrating with various AWS services such as ECR, ECS, and EKS. A2C generates CloudFormation template to configure required compute, network, and security infrastructure to seamlessly deploy containerized application in AWS. A2C creates CI/CD pipelines for Amazon DevOps services such as CodeBuild and CodeDeploy to build and deploy containers. If you have existing CI/CD tooling (for example, Azure DevOps, and Jenkins), then you can integrate A2C provided deployment artifacts into your existing CI/CD workflows. https://aws.amazon.com/app2container/faqs/

Claim 14	Accused Instrumentalities
	Q: How do I get started? Once the prerequisites are met and the App2Container tool is installed on the application server, use the following sequence of commands to containerize the application. • Init: Configures the tool with your AWS profile and other related parameters. • Inventory: Identifies all running applications that are supported by A2C. Choose an application to proceed. • Analyze: Performs detailed analysis to identify application artifacts, third-party dependencies, network ports and configuration files, and generates a configuration file to represent the gathered information. You can make additional changes to the configuration file (for example, exclude or include any additional files or directories). • Containerize: Generates a dockerfile and container image for the analyzed application. If needed, you can modify the A2C-generated dockerfile and rebuild the container. You can also perform local testing of the containerized application using docker run command before deploying to AWS container services. • Generate Deployment: Registers the container image with Amazon ECR, creates ECS task definitions, and Kubernetes deployment YAML file to deploy the containerized application to Amazon ECS or Amazon EKS. A2C generates CloudFormation template to create required infrastructure to deploy the containerized application. Optionally, it also generates AWS CodeBuild and CodeDeploy artifacts to enable repeatable way to build and deploy the containerized app. https://aws.amazon.com/app2container/faqs/ **Q: What artifacts are generated by App2Container?** App2Container generates the following artifacts for each application component: 1) Application artifacts such as DLLs configuration, 2) Dockerfile, 3) container image in ECR, 4) ECS Task definitions, 5) Kubernetes deployment YAML, 6) CloudFormation Template, 6) AWS CodeBuild and CodeDeploy resources. https://aws.amazon.com/app2container/faqs/